2012 IEEE Fifth International Conference on Cloud Computing

WIQ: Work-Intensive Query Scheduling for In-Memory Database Systems

Stephan Kraft*
SAP Research
Belfast, UK
stephan.kraft@sap.com

Giuliano Casale
Dept. of Computing
Imperial College London
London, UK
g.casale@imperial.ac.uk

Alin Jula
SAP Labs
Palo Alto, CA, USA
alin.jula@sap.com

Peter Kilpatrick‡, Des Greer†
School of EEECS
Queen's University Belfast, Belfast, UK
‡*Email: p.kilpatrick@qub.ac.uk*
† *Email: des.greer@qub.ac.uk*

***Abstract*—We propose a novel admission control policy for database queries. Our methodology uses system measurements of CPU utilization and query backlogs to determine interference between queries in execution on the same database server. Query interference may arise due to the concurrent access of hardware and software resources and can affect performance in positive and negative ways. Specifically our admission control considers the mix of jobs in service and prioritizes the query classes consuming CPU resources more efficiently. The policy ignores I/O subsystems and is therefore highly appropriate for in-memory databases. We validate our approach in trace-driven simulation and show performance increases of query slowdowns and throughputs compared to first-come first-served and shortest expected processing time first scheduling. Simulation experiments are parameterized from system traces of a SAP HANA in-memory database installation with TPC-H type workloads.**

***Keywords*-performance; database management; query interference; admission control; scheduling;**

I. Introduction

Cloud computing offers the ability to access large pools of hardware resources at low economic costs. In combination with the parallelization capabilities of multicore servers, this has promoted in recent years the migration of computationally-intensive applications from internal data centers to the cloud. For example, enterprises are increasingly using cloud infrastructures to access or host data analytics and business intelligence services [1]. The workload of these services often consists of long batch jobs processing very large data sets recording sales and customer information. Since longer execution times correspond to increased costs for the service provider, hosted services are requested to deliver maximum performance from their software stack. For this reason, in-memory databases have recently emerged as a mainstream technology to maximize responsiveness of data-intensive applications [2]. By manipulating large data sets directly in memory, and therefore omitting time-consuming disk operations, in-memory databases execute data-intensive operations in a fraction of the time of traditional databases.

This paper is motivated by the problem of defining effective resource management methodologies for in-memory database systems. Our main contribution lies in the analysis and optimization of a novel admission control policy for query execution called *work-intensive query first* (WIQ). We prove by trace-driven simulation that this policy is more efficient than popular policies such as first-come first-served (FCFS) or shortest expected processing time first (SEPT).

A challenge that arises in defining a good admission control policy for database systems is query interference. Recent work has highlighted the tendency of queries to interfere with each other when concurrently accessing a database [3]. Such interference is not limited to heightened contention at the cores, but involves also delays due to memory access or due to the internal characteristics of the software. For example, a decisive factor for database performance is the maximum number of queries that are allowed to run concurrently in the system [4], [5]. For example, choosing a small concurrency level results in under-utilized resources and low throughputs. Conversely, large concurrency levels may induce excessive contention delays for cores or exhaustion of memory resources. Furthermore, observed performance can depend on the specific job mix in service, since the database query planner organizes query executions in a sequence of consecutive or parallel subtasks that are spawned synchronously, thus suddenly increasing core contention for the other query types. Query plans usually vary for queries of different types resulting in different parallelization levels decided at run-time and that may also change during the lifetime of the query. Additionally, query completion times depend on the amount of query interference inherent in the query mix executing at the same time. For example, query interference can decrease completion times when queries share data loaded into buffers. Conversely, completion times are impacted negatively if queries compete for table locks.

The WIQ admission control policy, proposed in this paper, introduces a novel scheduling policy for query admission control. The policy ignores I/O subsystems and therefore is highly appropriate for in-memory databases. The policy uses minimal system measurements to determine query classes, i.e. query types, that consume CPU resources more efficiently in relation to the average backlog of requests they impose in the system. In essence, the policy maintains a table that characterizes the interference between pairs of classes

*S. Kraft is also affiliated with Queen's University Belfast.

978-0-7695-4755-8/12 $26.00 © 2012 IEEE
DOI 10.1109/CLOUD.2012.120

and prioritizes query classes that are able to use resources more efficiently, typically at the expenses of other query classes. The metric used to quantify class interference is obtained by relating the system observations to the theoretical behavior in the idealized case without admission control or software threading limitations. Our main result here is to prove theoretically that, in the idealized case, utilizations and backlogs need to stay in fixed proportion across classes at the instants when queries arrive to the cores. This enables the identification, in the real system, of classes that are able to use resources better than expected.

The WIQ policy is validated using TPC-H [21] type workloads and system traces from a commercial solution, SAP HANA [2], a product that aims at improving the performance of complex business intelligence workloads using the in-memory database technology. Although our work is illustrated on a specific product, the proposed methodology is rather general and may be applied to other in-memory database solutions. The remainder of the paper is organized as follows. Section II gives an overview of related work and Section III introduces the proposed resource management methodology. In Section IV we present the trace-driven simulation model and validate its quality in Section V. The resource management methodology is evaluated in Section VI. Section VII offers summary and conclusions.

II. Related Work

There is a long history of studying scheduling algorithms for parallel processing [6] and database transactions [7]. Usually these techniques require information of application internals. More recently interposed schedulers have been proposed maintaining blackbox views of applications. Approaches include an external admission control for e-commerce web traffic with shortest job first (SJF) scheduling [8]. In [4] database transactions are externally queued in an effort to control the level of concurrency in the database. None of these techniques specifically consider job interference in scheduling decisions.

Multiple Query Optimization (MQO) techniques account for job interference by exploiting database application implementation details and query structures to optimize query plans [9]. Closer to our work are the following methods maintaining blackbox views of the database. The authors in [3] investigate interactions between batched queries and propose algorithms for interaction-aware scheduling based on experimental sampling and regression techniques. The interference between query pairs is studied in [10]. Contrary to us these approaches do not consider resource usage. In [11] authors rely on measurements of disk accesses to determine interference of query pairs. We instead use CPU measurements to built our model.

Simulation models for performance evaluation of database designs have been proposed as early as the 70's [12], but did not prevail as a commonly used tool of database administrators. A probabilistic model for analysis of transaction scheduling in distributed real-time databases is presented in [13]. Closer to us, authors in [14] and [10] mention a simulation framework for evaluation of query scheduling disciplines. More recently, a simple mathematical simulation model for batched queries is presented in [15]. None of these tools consider the parallelization of queries into subtasks or are parameterized from system traces.

III. Resource Management Methodology

A query that executes on a database system has to compete with concurrently executing queries for core, memory and software resources. Performance modeling theory has developed over the years analytical tools to describe contention for core access using stochastic processes, e.g., product-form queueing network models [16]. This theory explains resource contention between different jobs when these are served under processor-sharing scheduling, an idealized version of round-robin where the quantum of time that each job receives is infinitesimal small. Product-form queueing network models are an appropriate abstraction to describe core contention, but they place some limitations on the analysis of real-world systems. For example, queries submitted to a database server may also experience blocking delays caused by constraints on the maximum software threading level. Thus, a query class with high parallelism level, but low core processing requirements, may still block the execution of other query classes if it saturates the threading limit constraint. These blocking delays add to the core contention overheads and can be difficult to model. We shall refer to such delays as *interferences* and, with the aim of defining a new admission control policy, we aim at characterizing and exploiting interferences between query classes.

We give in this section two contributions. First, we define a metric for characterizing the interference between any pair of query classes. Next, we propose the WIQ admission control policy to leverage this information for better resource management for in-memory databases. Specifically, we follow this approach:

1) We first define a simplified queueing model for the performance of the database server in the idealized case where there is core contention, but *no* interference, between query classes. Thus, this system ignores aspects such as task forking and admission control. We prove that, in this idealized model, a certain relationship must exist between utilization and queue-lengths (i.e., backlogs of queries) seen at the cores at the instant of admission into the system of a new query by the planner. We shall refer to this relationship as the *no-interference baseline*.
2) We then focus back on the real system. Based on the idealized result, we define a metric that quantifies how much the behavior of queries in the real system

deviates from the no-interference baseline. This will be used as a metric for admission control.

3) Finally, we define the WIQ admission control policy that tries to prioritize query classes that are expected to maximize core utilization in presence of interference. Simulation results demonstrate that the proposed policy outperforms policies such as SEPT.

A. No-Interference Baseline

We begin by defining an idealized model for the database server performance by considering a closed queueing network composed by M processor-sharing queues, modelling the cores, and a delay station, modelling the time taken by the scheduler to issue the next query after completion of the previous one. This time, called *think time*, is zero if the scheduler queue contains jobs, otherwise equals the residual time before a client issues the next request to the database. We denote by Z_c the mean think time of a class-c query. We ignore in this idealized model aspects such as maximum threading levels, thread forking, and latency for memory access. In order to mimic the behavior of threads, we require each job to perform on average a number visits to each core, before completing, such that the total processing demand $D_{c,i}$ of class-c queries, summed across visits, placed on each core i is identical to the one in the real system for each query class. Since cores are assumed identical, we define $D_c = D_{c,1} = D_{c,2} = \ldots = D_{c,M}$, where M is the number of cores. Let U_c be the utilization of class-c queries averaged across cores. Also, let A_r^c be the mean number of jobs of class r executing at a core, averaged across the cores, as observed by a query of class c at the instant it is issued by the query planner. We prove the following result.

Theorem 1. *Under the above assumptions, utilizations and mean arrival queue-lengths satisfy the following relation*

$$\frac{U_r}{U_c} = \frac{A_r^c}{A_c^r}$$

for all classes $c, r = 1, \ldots, K$ *where* $c \neq r$.

Proof: Under the given assumptions, the model under study is a product-form closed queueing network, possibly with class switching [16]. The population of the model corresponds to the population of cycling jobs. Let G be the normalizing constant of the equilibrium state probabilities of the underlying Markov chain model that describes the stochastic evolution of the network. Also let the notation $G_{c,r}^+$ indicate the normalizing constant for a related model where we remove a job of class c and a job of class r from the network, but increase the number of cores by one. Let also G_c and G_r be the normalizing constants of models with a job less of class c and r, respectively. For the model under study, it is established that [17]

$$A_r^c = D_r \frac{G_{c,r}^+}{G_c}, \quad A_c^r = D_c \frac{G_{c,r}^+}{G_r}$$

Therefore

$$\frac{A_r^c}{A_c^r} = \frac{D_r}{D_c}\frac{G_r}{G_c}$$

Recall that the class-c and class-r throughputs, respectively denoted X_c and X_r, are defined in terms of normalising constants as

$$X_r = \frac{G_r}{G}, \qquad X_c = \frac{G_c}{G},$$

then

$$\frac{A_r^c}{A_c^r} = \frac{D_r}{D_c}\frac{G_r}{G}\frac{G}{G_c} = \frac{D_r}{D_c}\frac{X_r}{X_c} = \frac{U_r}{U_c}$$

where we used in the last passage that, by the utilization law [18], $U_r = X_r D_r$ and $U_c = X_c D_c$. This completes the proof. ■

Based on the last result, it is natural to define the no-interference baseline as the metric

$$\eta_{r,c} = \frac{U_r}{U_c} - \frac{A_r^c}{A_c^r}$$

which for the idealized system satisfies $\eta_{r,c} \equiv 0$ for every possible choice of classes r and c. A discussion of this metric is provided in the next subsection.

B. Interference in the Real System

Using trace-driven simulation and the measurement discussed in the next sections, we tested if the observed utilization levels and arrival queue-lengths in a SAP HANA DB, a real-world in-memory database, satisfied the no-interference baseline. Without loss of generality, the queue-length values for this system are intended to be of threads instead of jobs. Our empirical observation is that, in general, the no-interference constraint is not satisfied, except in the obvious case $r = c$, and different pairs of classes (r, c) exhibit different degrees of deviation from the idealized constraint. Our proposal is to interpret this deviation as a measure for class interference, i.e., the delay occurring between classes that cannot be explained by core contention. We have in particular three cases:

- Case $\eta_{r,c} > 0$. This corresponds to the situation where

  $$\frac{A_c^r}{U_c} > \frac{A_r^c}{U_r}$$

  For example, this may be due to class c placing a higher A_c^r value than expected. Thus, class c consumes the threading limit constraint per unit of utilization *relatively* more than class r. Under this situation, it seems more efficient to execute class r queries instead of class c queries. The same conclusion is obtained after rearranging the terms as

  $$\frac{U_c}{A_c^r} < \frac{U_r}{A_r^c}$$

  which suggests that class r produces more work (higher utilization) per thread queued at the core. Thus, in

relative terms, we say that class r is more *work-intensive* than class c per thread queued at the core.
- Case $\eta_{r,c} < 0$. This is similar to the previous case and corresponds to a situation where class c utilizes more efficiently resources with respect to class r per thread queued at the core.
- Case $\eta_{r,c} = 0$. The two classes utilize resources in a fair manner consistent with the no-interference model.

Before proceeding further, it may be useful to observe that the $\eta_{r,c}$ metric has also the advantage of accounting indirectly for the response time of each class via the arrival queues A_r^c and A_c^r. It is well known in queueing theory that, in absence of prioritization mechanisms, such queues are the main determinants of end-to-end response times [18]. Therefore, our metric does not only consider utilization work, but indirectly accounts also for the time each requests spends in the system.

C. Work-Intensive Query First Policy

Based on the observations in the previous subsections, we propose a new admission control policy for in-memory databases called *work-intensive-query first* (WIQ). For a system with K classes, the policy relies on offline definition of a matrix $\boldsymbol{\eta} = [\eta_{r,c}]$, where element $\eta_{r,c}$ represents the deviation from the no-interference baseline for the pair of classes in row r and column c, $r, c = 1, \ldots, K$. Such table is built in this paper using measurements and the trace-driven simulation model defined in Section IV and thus requires the availability of trace data. The WIQ admission policy attempts to optimize system performance by selecting, among the jobs queueing in the system admission queue, the earliest arrived query of class r^* in the admission queue such that

$$\sum_{c=1}^{K} \eta_{r^*,c} Q_c^{dec} \geq \sum_{c=1}^{K} \eta_{r,c} Q_c^{dec}, \qquad \forall r \neq r^*$$

where Q_c^{dec} is the *current* number of class c jobs in service at the cores at the instant when the scheduler needs to take the admission control decision. (Note that $Q_c^{dec} \neq A_c^r$ since the latter is instead the *long-term average* for the number of threads observed when a class r query is admitted to service.) Based on the observations in the previous section, we interpret the WIQ scheduling decision to prioritize queries from classes that on average provide best expectations of being highest work-intensive per thread queueing at cores.

IV. Trace-driven Simulation Model

We defined a simulation model for performance evaluation of the proposed admission control policy for in-memory database systems. The simulation model is helpful for what-if comparison with other admission control policies and provides a testbed for computing the $\boldsymbol{\eta}$ matrix defined by WIQ. The model implementation extends JINQS [19], a library for simulating multiclass queueing networks, and is parameterized solely from measured system traces. Maintaining a blackbox view of the database internals is advantageous since such details may not always be available, usually complicate the model, and can require modeling changes once the application implementation is altered.

We consider a workload of clients interactively accessing hosted database services and use the closed queueing model depicted in Figure 1 for performance evaluation. The model specifically captures delays due to contention for hardware and software resources. Database performance degradation due to hardware contention has been mainly linked to storage resources in previous work [11]. However, the absence of time consuming disk I/O operations on in-memory platforms allows us to focus on modelling contention delays for cores. We represent available system processors with a network of processor-sharing queues in the model.

In multicore environments, databases can efficiently utilize parallel processors by incorporating concurrency into query plans. Parallelizing large portions of query tasks may result in contention for threads allocated to the application. We model the maximum threading limit with a finite capacity region (FCR) [18]. A FCR is a subnetwork with a constraint on the maximum number of jobs that can simultaneously visit it at a given time. Arriving jobs are added to an *AdmissionQueue* if available capacities are exhausted. No jobs are dropped. The *AdmissionQueue* is governed by a configurable admission control policy, e.g., non-preemptive FCFS, SEPT, etc. When the FCR capacity is exceeded, upon departure of a job from the FCR to the outside, the next job to be admitted is chosen within the *AdmissionQueue* by the admission control policy.

Consider the jth query instance, i.e. job, requesting access to the FCR. In the real system, this corresponds to the jth query received by the database from the clients. In the simulation, we describe this job by a class $k(j)$, which describes the type of operation performed, and by an iteration number $i(j) \leq I(j)$, where $I(j)$ represents the maximum number of cycles job j performs in the FCR before leaving. This corresponds to the number of processing phases the query undergoes before completing. Thus, a cycle corresponds to a single execution round at the cores. We define the response time for job j as the difference between time of its completion at the FCR and the time of its arrival at the admission buffer.

A. Fork and Join Elements

The parallelization of queries is captured with fork and join elements. The *JobForkStation* element forks a job into a number of *tasks* and essentially models a client query being broken up into smaller tasks by the query planner. We assume cores to have identical processing speeds and jobs to have equal routing probabilities to each core. The number of tasks forked for job j at cycle $i \equiv i(j)$ is denoted by $f_{j,i}$ that is a positive integer. Similarly, the time demand placed by each task at the core at cycle i is indicated with $d_{j,i}$. After

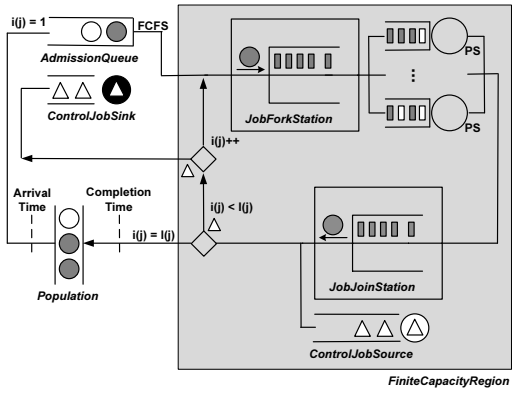


Figure 1. Simulation Model.

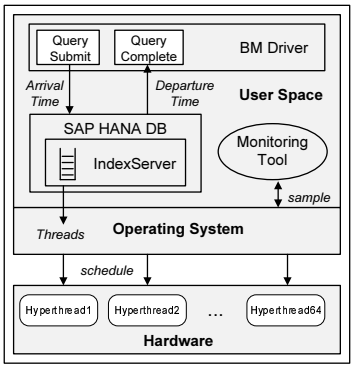


Figure 2. Reference System.

leaving the cores, all tasks are progressively accumulated by the *JobJoinStation* element until all $f_{j,i}$ tasks have arrived. At this point, they are merged back into job j which is then either moved to the next cycle $i(j)+1$ or departs the network if $i(j) = I(j)$. Thus, the time to complete cycle i equals the maximum residence time of the $f_{j,i}$ forked tasks into the queues modelling the cores.

B. Finite Capacity Region

The FCR models a constraint on the quantity of available software thread resources in the database application. We implement this restriction with a global counter tracking how many forked jobs have entered the FCR region. Jobs positioned in the *AdmissionQueue* are granted FCR access if the nominal job fork size, i.e. the fork count at iteration $i = 1$, does not exceed the available FCR capacity. That is, let

$$n_{fcr}(t) = \sum_{j \in \text{ jobs in FCR}} f_{j,1} \qquad (1)$$

be the number of jobs in execution inside the FCR at time t. Then the FCR imposes

$$n_{fcr}(t) \leq N_{fcr}, \quad \forall t$$

where N_{fcr} is the total FCR capacity, an input parameter of the model that limits the concurrency level inside the region. Tasks from different jobs concurrently executing on the cores may place different service demands and are therefore likely to complete execution at the cores at different times and with different orders with respect to the arrival order. Our model introduces control signals that are triggered as soon as a forked task arrives at the *JobJoinStation* element. The control signals decrement the FCR counter and are routed from the *ControlJobSource* to the *ControlJobSink* elements in the model as shown in Figure 1.

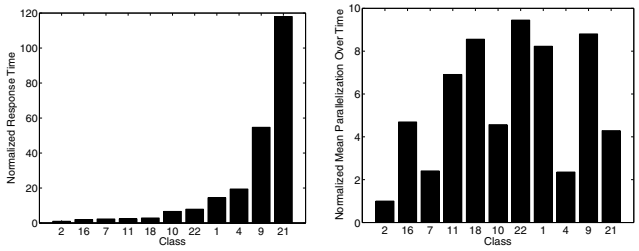
V. Simulation Model Validation

A. Reference System

For validation of our methodologies, we used a dataset of benchmark experiments performed on a four socket IBM X5 server with Intel Nehalem 7560 chipset and 1TB of RAM. This dataset was obtained and studied in [20]. The server features 32 physical cores, each clocked at 2.27Ghz with hyperthreading enabled, thus it can leverage on 64 hyperthreads. (In what follows, we use the terms core and hyperthread interchangeably.) The in-memory database used in the experiments is SAP HANA DB in the General Availability (GA) version 1.00. The high-level architecture of this system is illustrated in Figure 2. Once a database client submits a query request, it first enters the internal admission buffer of the *IndexServer*. This software component maintains the query execution control and is responsible for planning and management of query execution steps. Query planning is essential to database performance as complex queries may require multiple separate steps to complete (selects, joins, ...). The *IndexServer* decides this parallelism level in such a way to utilize cores efficiently. A configurable maximum threading level is maintained in order to control the parallelism level. Depending on the job mix currently in execution and on task parallelization levels when the query plan is completed, queries issued to HANA that cannot be served by the available threads wait in the admission buffer.

B. Workload

The workload is based on the TPC-H [21] benchmark which emulates workloads of online analytical processing (OLAP) systems. This benchmark defines $K = 22$ query classes. For amplification of the analytical workload, additional columns were added to the database tables *lineitem* and *orders*. The database was populated with a 1.3TB dataset (80GB working set in memory after HANA DB compression). Benchmark experiments were executed with different numbers of clients. In single client mode, denoted CON1 (concurrency level 1), the benchmark driver at any time has only a single outstanding query (i.e., job) into the system. In concurrent user mode, multiple client streams submit jobs so that job response times are affected due to contention. A client randomly chooses the class of the next job and upon its completion submits the subsequent job without delay. Two scenarios with 8 and 16 concurrent clients, respectively denoted CON8 and CON16, were available in the



(a) Response Times (b) Parallelization Levels

Figure 3. Single Client Measured Response Times and Parallelization Levels for Query Subset.

dataset. All benchmarks are executed over a period of 1 hour during which timestamps were recorded for query arrival and departure events. The dataset reports thread status, i.e., affinity to a certain core, in sampling intervals of $0.2s$. This sampling rate may appear coarse grained for reporting core thread affinities, but is appropriate for the workload under study as it consists of queries with mean processing times multiples larger than the sampling rate.

The workload contains a set of database queries with quite diverse characteristics. Due to the large number of classes and some long execution times, the measurements for some classes were limited to a few points. We determine the confidence of the measurements using 95% confidence intervals (CIs) and compute the CI widths as relative values to the mean. Figure 3 reports measurements for classes producing the reasonably low CI widths ranging in $[24\%; 130\%]$. Measurements are normalized by the values of class 2.

Figure 3(a) conveys that the range of query response times is large. A set of short running classes have small response times close to the base class. Classes $\{10, 22, 1, 4\}$ can be characterized as medium length runners and have response times roughly up to 20 times larger than class 2. Finally we record the two significantly longest running classes $\{9, 21\}$. The diversity in workload classes is further highlighted by the measured parallelization levels. Figure 3(b) shows the mean parallelization levels over query processing phases as described in Section V-C and illustrates that parallelization levels induced by the query planner are unique for each class. Interestingly, there is no clearly visible relationship between query response time and level of parallelization.

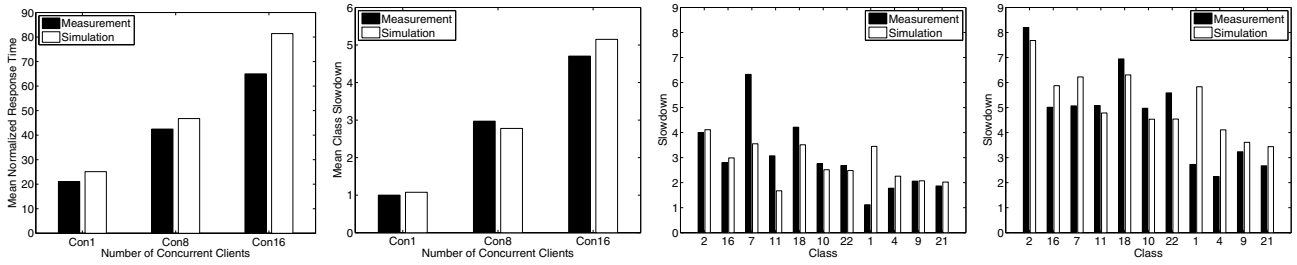
C. Validation

For validation of the proposed simulation model, we simulate the CON8 and CON16 scenarios and compare simulation results to measurements. The model input parameters job *fork size* and *service time* are computed from core thread affinity traces included in the dataset. In particular we use data from isolation runs, i.e. with only a single query in execution, where over the length of the run the total number of active threads pertaining to the query process are recorded. The amount of threads during query execution may change depending on parallelization levels in the query plan. We count each change in parallelization levels, i.e. number of active threads, as a separate processing phase of the query. In measurements we keep record of the number of active threads per processing phase and the length of time the processing phase lasted. In simulation the number of active threads in a given processing phase and the time interval the phase was observed are used as parameters for fork sizes $f_{j,i}$ and service times $d_{j,i}$, respectively. The measured query processing phases map to the modeling concept of job iterations, denoted $i(j)$. Furthermore, our reference system maintains 64 cores and consequently our model comprises 64 processor sharing queues. Additional input parameters for the model are job think times, job population size (N), and FCR threading limit (N_{fcr}). For think times we use measured traces from CON8 and CON16, respectively. Populations are estimated from measured throughput and response times using Little's law [18] ($N = 41$ for CON8 and $N = 60$ for CON16). The FCR parameter N_{fcr} was unknown, there it was chosen to the value producing similar mean job arrival queues in simulation compared to the CON8 and CON16 data ($N_{fcr} = 180$ for CON8 and $N_{fcr} = 275$ for CON16).

Our simulation experiments show that the model captures measured contention delays reasonably well. Figure 4(a) reveals that simulation tends to overestimate mean response times. Define the slowdown S_k of class-k queries as the ratio between that class response time and its demand D_k, estimated as the response time in the CON1 experiment. As shown in Figure 4(b), the mean class slowdown is matched very accurately: We record simulation errors for CON8 and CON16 of 0.06 and 0.10, respectively. To better understand the source of the per-class errors, Figures 4(c)-(d) compare per-class slowdowns from measurement and simulation. The model accurately captures variances in slowdown values and closely matches most measurements. Class 7 appears to be difficult to model in CON8. Simulation data of Class 1 overestimates measurements in both CON8 and CON16 scenarios. Considering the complexity of the model topology and the total of 22 considered classes, low deviations from measurement appear of sufficient quality for the purpose of defining a validation environment for the WIQ admission control policy.

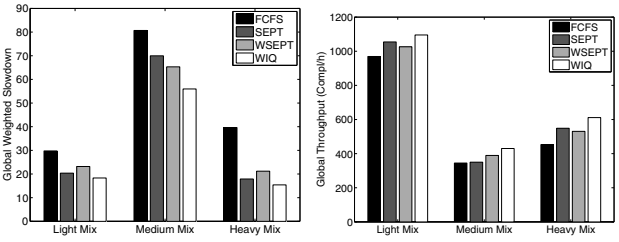
VI. EVALUATION

We now evaluate the performance of the proposed resource management methodology. We define three custom workload mixes and use our simulation model for a comparative study of job admission policies. In simulation the job scheduling is performed in the *AdmissionQueue* element depicted in Figure 1. In order to include realistic think times the model is parameterized with measured traces from CON16. We quantify the performance using throughput and weighted slowdown metrics. Weighted slowdown explains how much the slowdown of a particular class affects the



(a) Response Time (b) Mean Slowdown (c) Class Slowdown CON8 (d) Class Slowdown CON16

Figure 4. Mean Response Time and Slowdown in Measurement and Simulation.



(a) Slowdown (b) Throughput

Figure 5. Global Slowdowns and Throughputs in Simulations with First-Come First-Served (FCFS), Shortest-Expected-Processing-Time-First (SEPT), Weighted-Shortest-Expected-Processing-Time-First (WSEPT), and Work-Intensive-Query First (WIQ) Policies.

workload mix and is defined as

$$W(S_k) = S_k \frac{X_k}{X}, \tag{2}$$

where X_k and X describe the class k and total throughputs, respectively. Thus low $W(S_k)$ values are preferable.

Workload. For ease of interpretation of the experiment we consider a subset of six classes chosen from the reference workload. In particular we use classes $\{16, 18\}$, $\{1, 4\}$, and $\{9, 21\}$ to represent short, medium, and long running query types, respectively. We regard three workload mixes with a population of $N = 200$ jobs denoted *Light*, *Medium*, and *Heavy Mix*. The mixes are defined by the probability vectors $[0.6, 0.2, 0.2]$, $[0.2, 0.6, 0.2]$, $[0.2, 0.2, 0.6]$ specifying how many jobs of $[short, medium, long]$ running types are in the respective mix. We specifically chose workloads with high variance in job service demands. For example, Figure 3(a) reveals that class 21 response times in CON1 are roughly 8 times larger than class 1 and 60 times larger than class 16 response times.

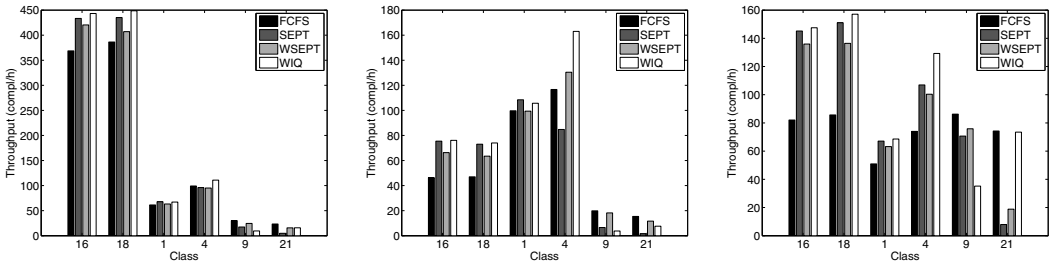
Admission Control. We compare the WIQ policy to three other policies that do not consider query interference and the job mix in service: FCFS, SEPT, and the custom extension Weighted-Shortest-Expected-Processing-Time-First (WSEPT). In related work, prioritizing short running jobs was applied successfully as admission control policy [8]. While SEPT prioritizes job classes with smaller CON1 response times, WSEPT factors in also the mean class parallelization level over processing phases. The WIQ policy uses the $\boldsymbol{\eta}$ matrix built from CON16 measurements.

Slowdown Result. Figure 5(a) shows the global weighted slowdown across classes and reveals that WIQ produces the lowest slowdowns for all workloads. SEPT is competitive only in the Light and Heavy Mixes. Considering the query interference in admission decisions appears to be most effective in the Medium mix. This also is the only case where WSEPT produces lower slowdowns than SEPT. Using FCFS in the admission buffer generates the largest slowdown values throughout all simulation runs.

Table I
MEASURED MATRIX $\boldsymbol{\eta}$ FOR CUSTOM WORKLOAD MIX.

	Class						
Class	16	18	1	4	9	21	Σ_{row}
16	0	0.1	0.1	0.3	0.5	0.4	1.4
18	-0.3	0	0.2	-1.5	1.0	0.8	0.2
1	-0.5	-0.3	0	-2.4	1.1	0.7	-1.5
4	-1.9	0.6	0.7	0	1.0	0.9	1.3
9	-25.2	-9.3	-9.4	-19.1	0	-1.0	-63.9
21	-9.4	-4.8	-1.8	-13.9	0.6	0	-29.4

Throughput Result. The global throughput results presented in Figure 5(b) reflect the slowdown findings and show that WIQ increases throughputs compared to the other admission controls. It is interesting to see that WIQ beats SEPT in the Light Mix where largest throughputs would naturally be expected with highest priorities for short jobs. To better understand the effects of WIQ admission we study the per class throughputs illustrated in Figures 6(a)-(c). In the Light Mix, depicted in Figure 6(a), WIQ generates highest throughputs for classes $\{16, 18, 4\}$. Consequently the $\boldsymbol{\eta}$ matrix in Table I confirms these query types most efficiently use the processing resources: table-row sums are the largest. Conversely, the long running classes $\{9, 21\}$ appear to use the resource less efficiently and therefore receive lower priorities from WIQ. Figure 6(b) illustrates maximal throughputs for class 4 in combination with WIQ in the Medium Mix. This is expected as classes $\{1, 4\}$ constitute the majority of the population and we compute the highest $\eta_{r,c}$ values for class 4. Interestingly WSEPT is more successful than SEPT in the Medium Mix. The long running classes $\{9, 21\}$ achieve the best throughput performance with FCFS. However, in workloads with large deviations in job service requirements FCFS can significantly penalize short running jobs, e.g., if jobs with service times of only a few seconds need to wait for jobs with service times of multiple hours to complete. The simulation results for the Heavy Mix are shown in Figure 6(c) and reveal that with the exception of class 9 WIQ achieves the largest throughputs



(a) Light Mix (b) Medium Mix (c) Heavy Mix

Figure 6. Throughputs in Simulations with Custom Workloads and First-Come First-Served (FCFS), Shortest-Expected-Processing-Time-First (SEPT), Weighted-Shortest-Expected-Processing-Time-First (WSEPT), and Work-Intensive-Query First (WIQ) Policies.

for long and short running queries. We interpret this behavior with the low no-interference values for class 9 in $\eta_{9,c}$ that indicate this class does not make efficient use of the core resources. Therefore the WIQ policy assigns low priorities for this class and the rest of the workload mix appears to benefit from this decision. To verify that WIQ does not lead to the starvation of low priority job classes we studied the tail distributions of response times in simulation. The results of the complementary cumulative distribution function, not reported due to space limitation, revealed that probabilities for large response times in WIQ were significantly lower than for SFQ scheduling in all workload mixes.

Summarizing. Simulation results indicate that system performance can be increased with the proposed resource management methodology. Specifically we compare the WIQ policy to FCFS and SEPT and find that accounting for job interference in admission control and prioritization of work-intensive queries reduces global system slowdowns and increases total throughputs.

VII. Conclusion

In this paper, we have proposed a novel admission control policy for database queries. Our methodology considers the complex interactions between queries in execution and determines the query classes consuming resources more efficiently. The approach is validated in trace-driven simulation experiments and proves to increase performance compared to FCFS and SEPT policies. Simulations are parameterized with system traces obtained from the commercial in-memory database SAP HANA and TPC-H type workloads. In future work we plan to further evaluate the capabilities of the proposed technique and implement the admission control policy in an online database installation. Additionally we aim to dynamically built the $\boldsymbol{\eta}$ matrix at runtime instead of during offline training.

Acknowledgment

The work of S. Kraft has been partially funded by the EU/SAP Cumulonimbo project. The work of G. Casale is supported by an Imperial College Junior Research Fellowship. Thanks to Stephen Dawson for valuable comments.

References

[1] J. Schaffner, B. Eckart, C. Schwarz, J. Brunnert, D. Jacobs, and A. Zeier, "Towards analytics-as-a-service using an in-memory column database," LNBIP. Springer, 2011, vol. 74.

[2] F. Färber, S.K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: Data management for modern business applications," *SIGMOD*, vol. 40, no. 4, 2011.

[3] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala, "Interaction-aware scheduling of report-generation workloads," *VLDB Journal*, vol. 20, Aug 2011.

[4] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman, "How to determine a good multi-programming level for external scheduling," in *ICDE*, April 2006.

[5] A. Mehta, C. Gupta, and U. Dayal, "BI batch manager: A system for managing batch workloads on enterprise data-warehouses," in *EDBT*. ACM, 2008.

[6] B.W. Lampson, "A scheduling philosophy for multiprocessing systems," *Commun. ACM*, vol. 11, May 1968.

[7] R.K. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: A performance evaluation," *TODS*, vol.17, 1992.

[8] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel, "A method for transparent admission control and request scheduling in e-commerce web sites," in *WWW*. ACM, 2004.

[9] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," *SIGMOD*, vol. 29, May 2000.

[10] M.-C. Albutiu and A. Kemper, "Synergy-based workload management," in *VLDB PhD Workshop*, Aug 2009.

[11] K. O'Gorman, A.E. Abbadi, and D. Agrawal, "Multiple query optimization in middleware using query teamwork," *Software-practice & Experience*, vol. 35, no. 4, Apr 2005.

[12] F. Nakamura, I. Yoshida, and H. Kondo, "A simulation model for data base system performance evaluation," in *AFIPS*. ACM, 1975.

[13] Ö. Ulusoy and G.G. Belford, "A simulation model for distributed real-time database systems," in *ANSS*. IEEE, 1992.

[14] M.J. Carey and M. Livny, "Distributed concurrency control performance: A study of algorithms, distribution, and replication," in *VLDB*, 1988.

[15] M. Ahmad, S. Duan, A. Aboulnaga, and S. Babu, "Predicting completion times of batch query workloads using interaction-aware models and simulation," in *EDBT*. ACM, 2011.

[16] F. Baskett, K.M. Chandy, R.R. Muntz, and F.G. Palacios, "Open, closed, and mixed networks of queues with different classes of customers," *J. ACM*, vol. 22, no. 2, 1975.

[17] G. Bolch, S. Greiner, H. de Meer, and K.S. Trivedi, *Queueing Networks and Markov Chains*, 2nd ed. Wiley, 2006.

[18] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

[19] T. Field, "JINQS: An extensible library for simulating multiclass queueing networks, v1.0 user guide," http://www.doc.ic.ac.uk/~ajf/Research/manual.pdf.

[20] A. Jula, "Multicore scalability and NUMA effects on HDB with SAP-H data and TPC-H queries," SAP, Tech. Rep., 2011.

[21] "TPC-H: Transaction processing performance council." [Online]. Available: http://www.tpc.org/tpch/default.asp